OULUN AMMATTIKORKEAKOULU

Kien Pham Gia

A WORK TIME MANAGEMENT WEB APPLICATION USING GPS

A WORK TIME MANAGEMENT WEB APPLICATION USING GPS

Kien Pham Gia
Final projects
Spring 2023
Bachelor of Engineering, Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
ICT, Web Development option

Author(s): Kien Pham Gia
Title of the thesis: A Work Time Management Web Application using GPS
Thesis examiner(s): Lasse Haverinen
Term and year of thesis completion: Spring 2023 Pages: 46 + 2 appendices

The main purpose of this thesis is to report on the process and the result of a creation of a web application. This application is a combination of two main parts: the website for the employer to monitor the time checked in by the employee, and a mobile app created using React Native for the employee to check in and check out of the workplace.

Additionally, this thesis will not just go through the process but explain the thought process during the creation and the lesson learnt after finishing the project.

The result of this project is a working, functional website and a mobile application for small businesses and their employees to monitor, check in and check out working time using GPS to validate location upon check-in and check-out. Most of the goals set out at the beginning are satisfied. Further improvements in testing and features are planned and will be implemented in the nearest future.

Keywords: GPS, time monitoring system, react native, expressJS.

PREFACE

The basis of this thesis came from a conversation with a senior developer I met at a conference, he gave the idea for this application to improve my ability in both backend and frontend. The development of this application and thesis took place in Oulu, Finland.

Oulu,
Kien Pham Gia

CONTENTS

VOCABULARY

API – Application Programming Interface

Apps – Application

CSS – Cascading Style Language

HTML – Hypertext Markup Language

JS – JavaScript

NPM – Node Package Manager

SQL – Structured Query Language

UI – User Interface

GPS – Global Positioning System

EJS – Embedded JavaScript

1 INTRODUCTION

This thesis is to report on the implementation of a web application for small businesses to monitor/manage their employees. This application is a personal project for the author to learn/improve on their development of full stack web apps using React Native JS and Express JS.

The backend of this project will use Express JS for a simple API which will be made to easy expand in the future and MySQL for the database. The application has 2 different front-end, one using EJS as view for admins of the employer and one using React Native JS for employees to use. In the employer website, EJS file will be sent to the frontend to show the content of the website. Redis are used to initiate and store session cookie for persistence session for employee website. Library such as Datatable and chart JS also be used to show data such as employee/employer listing with pagination and search. In the employee React Native app, geolocation library in the react-native-community and Geocoding API from google will be used to obtain the location and address of the employee to check-in/check-out of the workplace.

This thesis report will show detailed information about the technology used in the process of making this application.

2 GOALS

2.1 Short Description:

This project is a small employee management system for small businesses that aim to replace physical timecards. It will have two main components; one is the mobile app for the employee which allows the employee to count the time using GPS, and the second is the website for the employer to monitor their employee. The estimated time for this project will be 250-270 hours.

2.2 Goals:

2.2.1 Employee applications:

- Employees should be able to sign in.
- Employees should be able to change their password.
- Employees should be able to check in at their workplace using GPS.
- Employees should be able to enter custom working time data in case not working at the workplace.
- Employees should be able to send an absent message.

2.2.2 Employer website

- Employers should be able to sign in.
- Employers should be able to create a new employee account.
- Employers should be able to check and see their employee working time.
- Employers should be able to see their employee profile (Full name, email, address, and other contact information).
- Employers should be able to see their employee working time both per day and per month.

3 TECHNOLOGY USED

This chapter will define the technologies used within the project.

3.1 NodeJS

'As an asynchronous event-driven JavaScript runtime, Node.JS is designed to build a scalable network application' (1).
The server-side execution of JavaScript code can be done using the Node.JS runtime environment. Due to its effective and scalable architecture, it gained popularity after its initial release in 2009. Node.JS has an event-driven, non-blocking I/O architecture, which enables it to handle many concurrent connections without obstructing the performance of other processes, in contrast to conventional server-side technologies like PHP or Ruby (1).

The V8 JavaScript engine, which is also utilized by Google Chrome, is the foundation upon which Node.JS is built. This engine converts JavaScript script into machine code, which the computer's processor subsequently runs. Node.JS can deliver quick and effective performance thanks to V8.
Although Node.JS is frequently used to create web servers and APIs, it may also be used to create desktop programs, command-line tools, and even robotics applications. The ease with which Node.JS can manage real-time, data-intensive applications is one of its main advantages. This is made possible by its event-driven architecture, which enables it to manage several connections at once and process data as it comes in rather than pausing to wait for it to load completely (1).

The extensive and vibrant ecosystem of NodeJS is an additional advantage. Over a million open-source packages and modules are accessible using NodeJS package management, NPM. Functionality can easily be added by developers to their applications because of this (1).
Node.JS has a wide range of scalability options, including running on a single server or in a distributed setting with the aid of tools like Kubernetes or Docker. Additionally, it contains an integrated clustering module that enables programmers to extend their programs horizontally and benefit from multi-core CPUs (1).

3.2 NPM

NPM or Node Package Manager is the default package manager that comes with Node.JS. NPM allows developers to access, use and submit to a library of millions of open-source packages and modules. These packages can be anything from providing a small utility solution to an entire framework or application. Using NPM requires developers to have a ‘package.JSON’ file which contains the metadata of the project and its dependencies. NPM then will read the ‘package.JSON’ file to install the required dependency using CMD or Linux terminal to ‘node_module’ folder. NPM does not just allow the installation of the newest version of a dependency but also any version required by the developer. NPM also allows the developers to manage, update, and uninstall packages (2.)

3.3 ExpressJS

Express is a minimal and flexible Node.JS web application framework that provides a robust set of features for web and mobile applications (3).
Express JS is known for its lightweight, simplistic, and minimalistic approach. Express JS has many key features, such as Routing, Middleware, Templating Engines, Error Handling, and Static file Serving. ExpressJS also allow integration with other Node.JS libraries like database integration with MySQL or authentication with passport(3.)
Express will be used as the main backend framework for this project. Express provides an easy and quick way to create a REST API which will be used to send and receive required data from and to employee applications, along with sending EJS and JS files to the frontend of the employer’s website.

3.4 MySQL

MySQL is an open-source relational database management system or RDBMS. MySQL uses structured Query language to manage data. MySQL can be used with a wide range of operating systems and programming languages. MySQL allows a wide range of data types, like string, int, DateTime, Boolean, JSON, blob etc... Data are stored in tables with multiple columns and rows which can be individually defined according to the need of the project. MySQL also allows the use of a variety of storage engines, which decide how data is stored and manipulated. InnoDB is the default engine since the release of MySQL 5.5.5 in 2010, replacing MyISAM (4.)

This project uses MySQL as its database. This project only requires a small number of simple tables, making MySQL an excellent choice for its speed and ease of use.

3.5 EJS

EJS is a simple templating language that lets you generate HTML markup with plain JavaScript. EJS allows JavaScript to be embedded inside HTML code with simple syntax. EJS can be used with ExpressJS which will be the main use case in this project to create a website for the user. With EJS, templates and partials can be used to reuse the same component as header or footer across multiple pages. EJS can easily be used with ExpressJS by simply defining the view engine with EJS. First, install EJS with NPM: 'NPM i EJS' or 'NPM i -g EJS' if developers want to install EJS globally. Then in app.JS, set view engine to EJS.'app.set('view engine', 'EJS')' (5.)

3.6 React Native

'React Native combines the best parts of native development with React, a best-in-class JavaScript library for building user interfaces' (6).
React Native is an open-source application framework for mobile created by Facebook. It allows developers to create applications for both iOS and Android with React and a single codebase. React Native allows the use of native UI components, hence making the application has the native look and feel (6.)
React Native will be mainly used for the employee mobile application, this allows a nice-looking application without using platform-specific language i.e., Java for Android, and Swift for iOS.

3.7 Expo

Expo is an open-source tool for building mobile applications using React Native which consolidates logic for both iOS and Android)(7).
Expo comes with a CLI (command-line interface) that allows developers to create, develop and test their React Native applications. Expo also comes with various pre-built components and APIs like camera, geolocation, and storage, allowing developers to use and test their application directly in the browser or an iOS/Android simulator. Another feature of Expo that makes the development

process easier is the ability to send the app using its own Expo app on mobile phones by simply scanning the QR code. Applications published using Expo can update using OTA or over the air without requiring another download, this allows fast and easy bug fixes and feature updates. Expo also offers Expo push notification, authentication, and distribution application for developers (7.)

3.8 Google Geocoding API

Google Geocoding API is an API provided by Google to retrieve addresses from coordinates or vice versa, this will be the main tool to retrieve locations (8).
Google Geocoding API can be used by enabling Geocoding API from Google Cloud Console. After enabling, developers can use their API to send HTTP requests to Google servers for data. Retrieved data can be in either XML or JSON format (8).
To use the API, developers send the HTTP request with the following form:

`https://maps.googleapis.com/maps/api/geocode/`*`outputFormat`*`?`*`parameters`*

`outputFormat` can be either JSON 'JSON' or XML 'XML'

`parameters` contain :

- Can be 'address=' for geocode, the street address element should be delimited by spaces (url-escaped by %20).

OR

- Can be 'latlng=' for reverse geocode lookup, retrieving the closest human-readable address.

AND

- 'key=' this for Geocoding API key.

Optional parameters for reverse geocode lookup:

- 'language': for the language of the return response, if not included, the response will use the preferred language stated in the header or the native language of the domains.
- 'region' for regional code, using ccTLD ('top-level domain') two-character value.
- 'result_type' to filter address, separated by a pipe | . (8.)

The response of the API comes with the format in Figure 1.

```
{
        “results” : [
                “address_components” : [
                        {
                                “long_name” : string,
                                “short_name” : string,
                                “type” : [ string ]
                        },
                        …
                ].
                “formatted_address” : string,
                “geometry” : {
                        “location” : {
                                “lat” : float,
                                “lng”: float
                        },
                        “location_type” : string,
                        “view_port” : {
                                “northeast” : {
                                        “lat” : float,
                                        “lng”: float
                                },
                                “southeast” : {
                                        “lat” : float,
                                        “lng”: float
                                }
                        }
                }
        ]
}
```

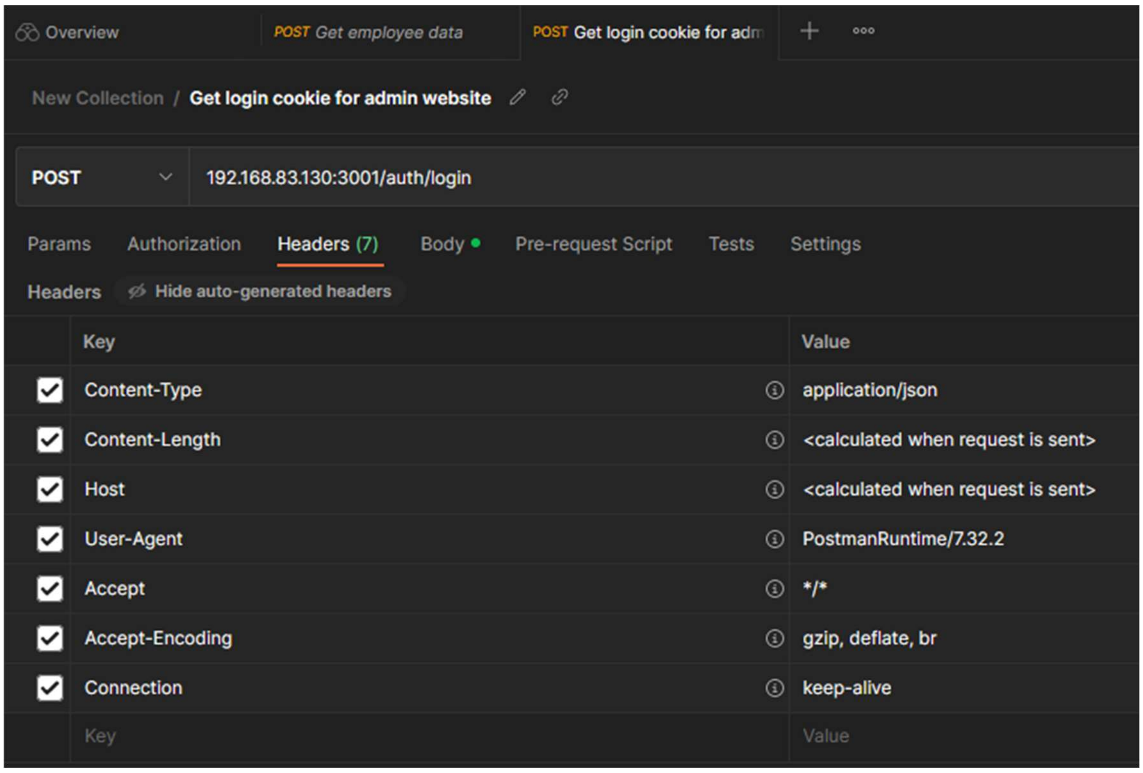
Figure 1, format of the received response from google geocoding API (8).

In this project, ‘formatted_address’ will be used.

3.9 Postman

Postman is used by developers to design, test, build and iterate their API (Application Programming Interface) (9.)

Postman has an easy-to-use, friendly UI (User Interface) that allow the ease of making HTTP request. Developers can modify everything in the HTTP request, from header, body, params, authorization, and even making scripts to test API(9). Figure 2 shows a part of Postman UI that shows how to configure the header part of the HTML POST request.

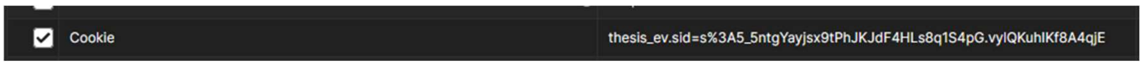

Overview
POST Get employee data
POST Get login cookie for adm
New Collection / Get login cookie for admin website
Params
Authorization
Headers (7)
Body
Pre-request Script
Tests
Settings
Headers
Hide auto-generated headers

POST 192.168.83.130:3001/auth/login

	Key	Value
✓	Content-Type	application/json
✓	Content-Length	<calculated when request is sent>
✓	Host	<calculated when request is sent>
✓	User-Agent	PostmanRuntime/7.32.2
✓	Accept	*/*
✓	Accept-Encoding	gzip, deflate, br
✓	Connection	keep-alive
	Key	Value

Figure 2. Header part of a POST request in Postman UI shown.

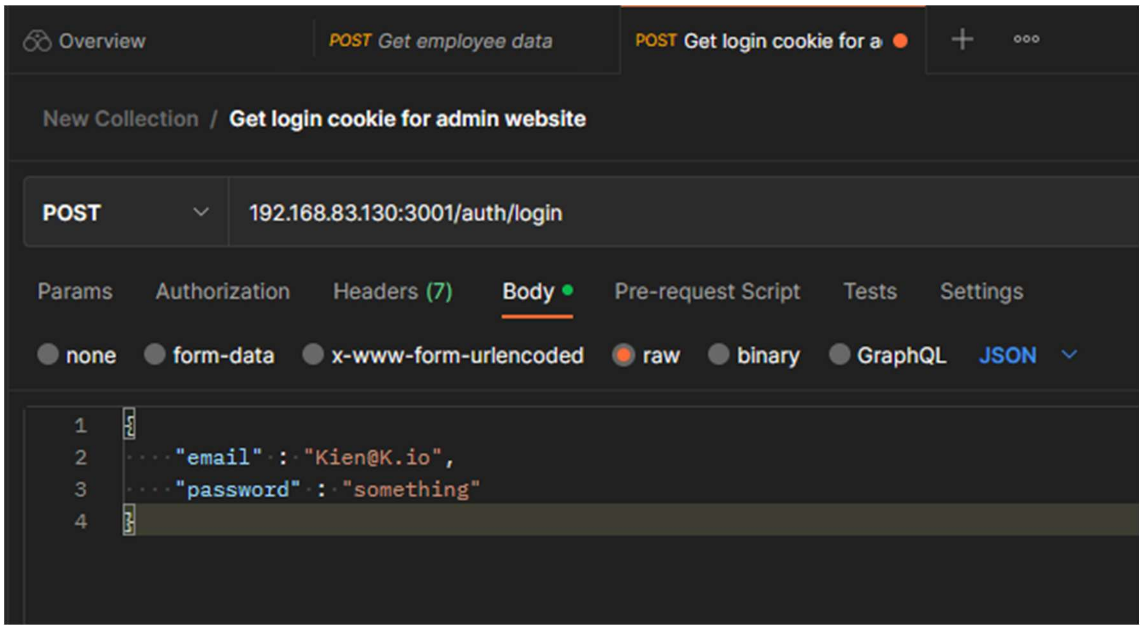
Postman auto-generated all the common headers but still allows the developer to add additional header elements with ease, in this case, cookies in Figure 3.



✓	Cookie	thesis_ev.sid=s%3A5_5ntgYayjsx9tPhJKJdF4HLs8q1S4pG.vylQKuhlKf8A4qjE

Figure 3. Cookie section of Postman UI.

Body:



POST 192.168.83.130:3001/auth/login

raw JSON

```
{
    "email" : "Kien@K.io",
    "password" : "something"
}
```

Figure 4. the Body part of a POST request in Postman UI shown.

Postman allows a variety of different body data formats, from raw, none formatted to in this case, JSON in the example in Figure 4.

3.10 HTTP Sessions

HTTP sessions are a mechanism for a web server to maintain and store user-specific data across multiple interactions with a web server. In other words, it allows user identity to be tracked across different pages and requests. The session is stored in a single-server, non-replicated persistent storage mechanism memory like cookie-based session persistent or file system persistent. HTTP sessions usually initiate when a client logs into an application. Once the server received the request, it will generate a Session ID. Subsequent request from the user will have the session ID included in the header, typically as a cookie from the browser or sent as a parameter in the URL. Using session ID, the server can associate user data, user preferences, and authentication information with the current session used by the user in the browser. HTTP sessions typically store user-specific data like session identifier or session ID, creation time, last accessed time, and other contextual info in memory (10.)

3.11 Redis

Redis or Remote Dictionary Server is an open-source, in-memory data structure store to be used as a real-time data store, caching, session storage, and streaming and messaging. Redis support a variety of in-memory data structure like string, hashes, lists, sets, sorted sets, streams and more. Redis data kept in memory allowing for quick access also can persist write to permanent storage, allowing for reboot or system failure. Redis also support server-side scripting with Lua with an extensive modular API for building custom extension (11.)

3.12 GPS

GPS or Global Positioning System is a system developed by the United State Department of Defence. This system uses a network of satellites orbiting around the Earth combines with a system of ground stations and user receivers to determine the location of the users. To GPS, a GPS receiver is required which is integrated into most if not all mobile devices. A GPS receiver uses the signal from the GPS to obtain a collection of data, such as location in the form of a set of coordinates and time. From the location and time data received from the GPS signal, the host machine can then calculate more information for the user like the velocity at which the user is travel at (12.)

4 DESIGN DECISIONS

4.1 Project architecture

Below in Figure 5 is the architecture of this project, as the diagram points out, there will be 2 different viewpoints, one for the employer and one for the employee. The one for the employer will be sent directly from the NodeJS server running ExpressJS as an EJS file to display. The one for the employee will be a standalone application written using React Native framework.
The employer website will have direct access to all the routes will authentication using a cookie.
The employee application will only have access to the '/employee' route using an API Key embedded into the config file of the application.
The database will run on MySQL database system using its default engine InnoDB. There will be 4 tables: admin, employee, 'employee_checkin_time' and settings. 'admin' and 'employee' tables will be responsible for storing data for admin and employee, in which password will be hashed with bcrypts. 'employee_checkin_time' table will be responsible to store each time employee check in or check out of their workplace. 'settings' table will store setting profile of which includes normal working time and workplace location.
Geocoding API from Google will be the only outside API being used for this project to send requests for location based on user latitude and longitude, which are obtained by using GPS.

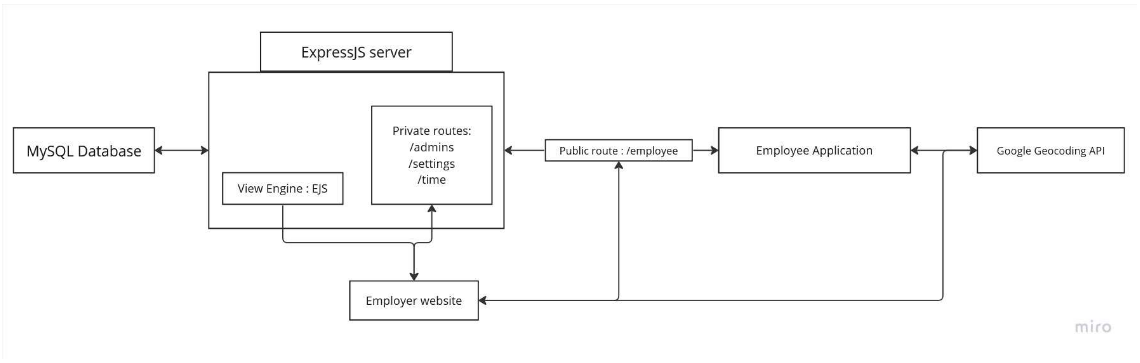


Figure 5. Application architecture illustration.

4.2 File Structure

4.2.1 Employee React Native App:

```
.
└ – – mobileApp
  ├ – – App.js
  ├ – – assets
  |    ├ – – adaptive-icon.png
  |    ├ – – favicon.png
  |    ├ – – icon.png
  |    └ – – splash.png
  ├ – – babel.config.js
  ├ – – components
  |    ├ – – changePassword.js
  |    ├ – – customTime.js
  |    ├ – – login.js
  |    └ – – mainPage.js
  ├ – – package.json
  └ – – package-lock.json
```

Figure 6. Employee application file structure.

The above Figure 6 shows the file structure of the mobile app which be used by the employee. The ‘App.JS’ file is the main master file. The assets like icons are stored in the appropriate named assets folder. Individual components like the login page, ‘customTime’ form and ‘changePassword’ form and main page are stored in the component folder which will be imported into the app.JS file to show at the appropriate time.

4.2.2 Employer Website:

```
└ – – server
  ├ – – app
  |   ├ – – passport
  |   ├ – – routes
  |   └ – – services
  ├ – – app.js
  ├ – – bin
  |   └ – – www
  ├ – – config
  |   ├ – – index.js
  |   └ – – schema.js
  ├ – – lib
  |   ├ – – db.js
  |   └ – – logger.js
  ├ – – package.json
  ├ – – package-lock.json
  ├ – – public
  |   ├ – – assets
  |   └ – – logos
  └ – – views
    ├ – – errors
    ├ – – index.ejs
    ├ – – layouts
    ├ – – login.ejs
    └ – – pages
```

Figure 7. Backend, and employee website.

In the above Figure 7 contains the backend API and employee website files.

In this file, there are 3 main folders: app, public and views, these folders contain the main part of this part of the project. The app folder contains 3 sub-folders, ‘passport’ containing the logic for checking and enriching the user credential. Routes responsible for handling different routes of the API. Services are for SQL calls to the database. Inside the public folder are the assets like logos and JS files. Views folders responsible for the employee website view file, written in EJS, these files are sent when an authorized user access certain routes including errors like 404 or invalid routes.

The other 3 folders: 'bin', 'config', and 'lib' responsible for the initialization of the server. The bin folder contains the script that would be run when the server initiates. Config and lib contain the information required by the application like MySQL credentials, ports, and console logging format.

4.3 Technology thought process and decision

4.3.1 Front-end:

The front end of this project makes use of 2 different technology, EJS and React Native JS.
For the employer side, the choice of EJS provides a quick and lightweight way to show data to the user. EJS in simple terms is an HTML file but can use JavaScript logic throughout to make the process easier. While it is entirely possible to do just an HTML file, EJS 'include' keyword allows the author to segment the common components to use across different pages to reduce repetition and allow a more consistent look. Another alternative would be using another front-end framework like React or Angular, however, the requirement for this part of the project is only for simply showing data and updating data using forms, the uses of EJS satisfied the requirement while being the simplest solution while still allow expansion if needed in the future.
For the employee site, with the need of being able for the employee to use their phone to check in to their workplace, the application needs to be coded in a way that it can work on 2 major phone operating systems: Android and iOS. To full fill this requirement, there are 3 main options, one is to use a web app, 2 is to use a native app, and the last is a hybrid app. The option of native apps requires the developer in this case, the author to write 2 different code bases, one for each platform, which complicated the process for a one-person team. The first option which is the traditional web-site used in a browser, comes with the benefit of the user can use the app without having to down-load or install additional applications directly on their device. However, the web app experience is also affected by which browser is used by the user, and web app tends to have features like buttons and menu bar that are difficult to use on a phone. The last option, a hybrid app is, as the name suggests, a between a traditional web app and a native app, in that it can be written with a single code base like a web app but can be translated into and used native feature (13.) It makes the process of making this application easier since while the targeted audience using a phone, the application does not require a large amount of device resources. From the pros and cons of said options, hybrid apps full fill the requirement the most. Within a hybrid app platform, React Native

not only full fill the technical requirement for the project, but it also full fills the personal requirement of this project for the author as it is written in JavaScript which is the language the author wanted to improve in.

4.3.2 Back-end:

The backend of this project uses Express JS and MySQL.

ExpressJS provides an easy and simple way to create an API for the uses of this project. There are other alternatives to ExpressJS like NestJS or Fasify, each comes with its pros and cons. However, with the timeframe and the scope of the project, which is for small businesses without too much traffic, the author decided on ExpressJS due to the experience of working with it in past projects.

MySQL is the database of choice for this project. There are other alternatives like PostgreSQL or MongoDB. However, the scope of this project is small, without needing complicated queries or databases, making MySQL for its ease of use, author familiarity and fast to set up while not compromising on speed compared to its competition in this scope of the project a perfect choice.

4.3.3 Authentication:

The project required the user to log in on both the employee side and the employer side. There are a lot of options to achieve this, in this section, we will evaluate and choose one from 3 options: cookie-based, token-based and OpenID. Cookie-based authentication uses cookies to handle user authentication. In this method, after the user posts the credential, the server will verify and create a session with the session ID. This ID is stored in the server and then send to the user using a cookie. Subsequent requests will require this cookie to work. Token-based works similarly but instead of saving the token in the server, the token will be saved in the local storage on the client side. Token-based authentication does not require the server to remember the interaction between it and the client. Each token is a self-contained string which includes enough data for the server to verify the user (14.) The last way or OpenID is using a third-party identity provider for the user to log in. However, since this application is meant for business use which might require a certain part of the application to run locally only on the business's server, this way of authentication would not work. Between token-based and cookie-based authentication, the main difference is where it is stored and how the server tracks user interaction (15.) In this case, the user needed to be tracked across different requests in a session which makes cookie-based authentication the better choice.

4.4 Frontend UI decision

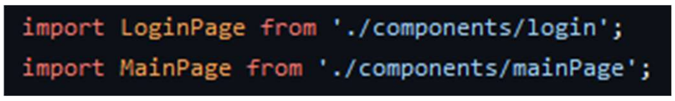
All frontends of both applications will be broken down into small components.

4.4.1 Employer website:

In this case, using EJS, the header, footer, and each modal used for forms are contained in a separate EJS file and stored accordingly to keep the design consistent across multiple pages.
Header and footer are stored in '/views/layout/header' and '/views/layout/footer' respectively.
HTML header where packages and script are included also will be done in a separate file to keep the file clean and easy to read and modify.
Each page will be shown by using the 'include' statement from EJS in a master EJS file.
After that, the master file will be included in the 'index.EJS' which also includes header, html header, footer, and logic to send JavaScript script according to which view file is sent to the frontend.

4.4.2 Employee application

This application is simplistic in terms of design. It only has 4 main pages: login page, main page, change password form and change custom time form.
However, it will still be broken down into components, and be included in the main 'app.JS' file.



```
import LoginPage from './components/login';
import MainPage from './components/mainPage';
```

Figure 8. an example of importing component to 'app.JS'..

5 IMPLEMENTATION

5.1 Employer Website and Backend API

5.1.1 Database Model

There will be 4 main tables: ‘admin’, ‘employee’, ‘employee_checkin_time’ and settings in this project database. Figure 9 is the visualisation of the database which will be used for this project. In Figure 20 in the appendix section, the author includes a schema of the database. The admins and employee table are used to store the admin and employee credentials and other data with the password hashed for security reasons. In ‘employee_checkin_time’, data involving time in and out of the employee are stored, ‘employee_id’ column is used to reference between this table and the employee table. The settings table will be used to contain other data like the location of the workplace to avoid hardcode the location in the application. The ‘working_time’ field of the table is configured to accept JSON files which allow for expansion in the future in terms of setting the exact working time for each role or payment per hour.

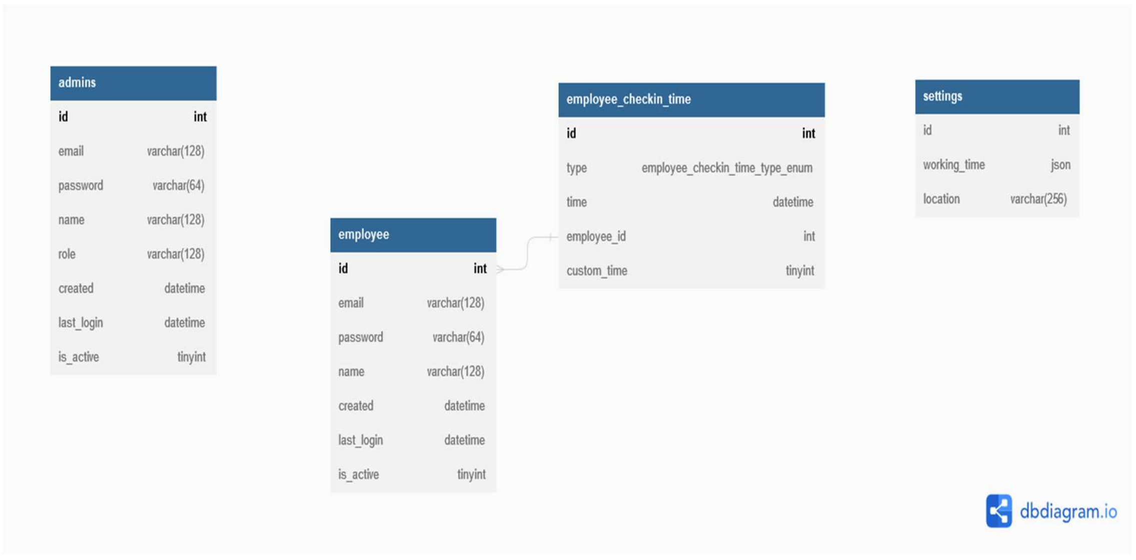


Figure 9. Visualisation of the database table.

5.1.2 Backend API

The backend API of this project contains 6 main routes: '/auth', '/admins', '/login', '/', '/employee', '/settings'. In these 6 routes, only '/login' is openly accessible, the rest requires either a session ID which is received by using login or an API key. When a user tries to access other routes without a session ID or API Key, they will be redirected back to login. When accessing '/login', the client will receive an EJS file which contains the form to enter their credential, if correct, the server will create a session and send a session ID back to the user which will be used to access other routes. In case '/login' does not work, credentials can be sent directly to '/auth/login' to retrieve a session ID .'/employee' route is the only other route that does not require a session ID but required an API Key, this route is used for the employee application in which the API is stored in. '/admins' and '/employee' and '/settings' are used to retrieve/ enter data related to the name of the respective route. More detailed documentation of the API can be found in the appendix.

5.1.3 Authorization

To authorize the user, in this project, Redis and passport middleware creates a session.
Users who access the website for the first time or logged out at the last accessed time, regardless of which route will be redirected back to '/login' if not authorized.
When redirected to '/login', the user will be prompted to input their account credential including email and password. After the credential is submitted, data will be sent to the server in the body using POST to '/auth/login'**.** The server then trimmed and lowercase the received email data to search in the database using 'adminServices'. If the email does not exist, then the server will send a message back to the client. If an entry can be found with the given email, then the password will be compared to the hashed password stored in the database. Credential is authenticated using passport Local Strategy to be serialised or deserialised. User session data is then stored using Redis store.

5.1.4 Listing users, changing passwords, and adding new users

Listing users

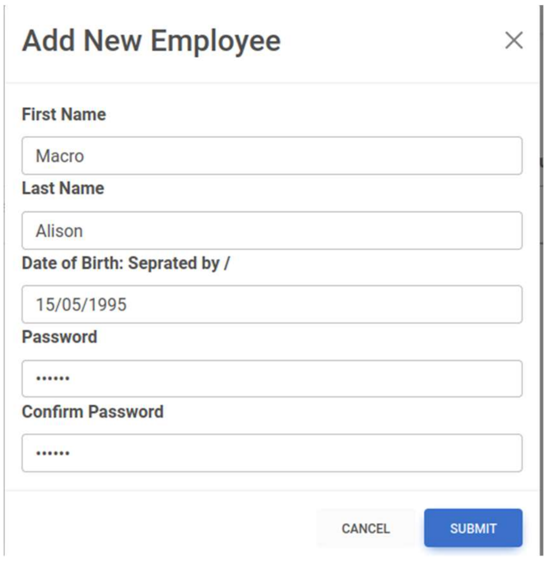
Employee and employer are both listed in '/employee' and '/employer' respectively. Data are paginated, sorted, and searched using the Datatable library. Data are sent from the database alongside the view file. Data is then parsed using the ‘for’ statement in the EJS file to be initiated into a Datatable.

Change the password, and Create New User

Both actions are done using forms. Both action forms are using bootstrap modals, and open using a button. Create new user button is above the listing table, allowing admins to add another admins user. The change password button for an individual user is on their page where more detailed data are shown. When data are submitted to the server, they will be validated. For changing the password, the old password will be compared to the hashed password in the database then the new password will be hashed and changed in the database, replacing the old one. For creating a new user, the email needs to be unique so no other entry with the same password should be found in the database.

5.1.5 Creating a new employee account:

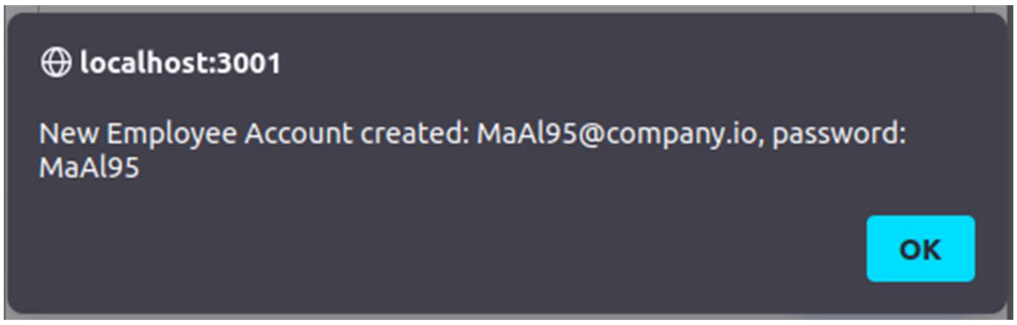
Creating new employee data are entered by a form which in a modal, opens using a button above the employee list. Data needed to create a new employee are their name both first and last, in separate input, and their date of birth, shown in Figure 10. The custom email is then created using the first 2 letters of both their first and last name with the last 2 digits of their year of birth. If an email already exists, a number will be added at the end and its numeric value will continuously increase until a valid email is found. The new account is then prompted as in Figure 11.



Add New Employee

First Name	Macro
Last Name	Alison
Date of Birth: Seprated by /	15/05/1995
Password	••••••
Confirm Password	••••••

Figure 10. Adds a new employee form.


localhost:3001

New Employee Account created: MaAl95@company.io, password: MaAl95

Figure 11. New employee account created.

5.1.6 Showing time data by day or month

First, the time data of the employee will be grabbed from '/employee/{employeeID}'. Data will be shown in the default setting of per day work time in a chart using the ChartJS library. How the data is grouped is selected using a dropdown which includes 2 options of day and month. This dropdown will trigger an 'onChange' function once it changes. Then the data will be calculated and added to a 'modifiedData' array which includes a date ('DD/MM/YYYY' for the day option and 'MM/YYYY' for the month option) and work time according to the data. When data is calculated, the chart will change accordingly.

5.2 Employee Application

5.2.1 Authorization

To authorize the user, the application after receiving the login credential from the login form will be sent a POST HTTP request to '/employee' with the data. After the server verifies the credential, the application receives a code 200 with a success message and the employee for future requests. 'isLogin' state then changes to true, allowing the user to access other pages in the application.

5.2.2 Location Checking

When a user clicks on the check-in or check-out buttons, the coordinate of the user will be determined using the geolocation library. The coordinate which includes latitude and longitude then send to Google geocoding API to get the location. The address in the 'formatted_address' in the response JSON received from Google API, then is sent to '/employee/time' using a POST request. After the address is sent to the backend, it would be compared to the set address of the workplace. If the address is correct, then the time which is sent along with the address will be logged into the system and counted for the employee.

5.2.3 Submit customised work time and change password

Users can send customised worktime and change their password in the application. Both actions are sent using a form. To submit customised work time, users need to fill in their start and end work time. The data will be sent in 2 separate HTTP requests, one for 'in' time and one for 'out' time, with the same format for normal check-in/check-out except with no location and 'custom_time' set to 1. To change the password, both old and new password need to be entered, then send to the server with POST protocol with the data a JSON format in the body. If the data is correct and the action succeeds, the user will be redirected back to the main menu.

6 TESTING METHOD AND RESULT

6.1 Testing

Due to time constraints, testing for this project are mainly using human interaction with the product and using postman for testing the API. The application is tested with wrong data input, trying to access routes which are supposed to be private. The application also stress-tested with a large amount of data in the database, created using a custom script in the MySQL database (Figure 12). ‘data’ table is a table that contains a set of valid data.

````
`
CREATE DEFINER = ‘root’@’localhost’
PROCEDURE  ‘INSERTRAND’ (IN numRows INT)
BEGIN
    DECLARE i INT;
    SET i = 1;
    START TRANSACTION;
    WHILE i <= numRows DO
    INSERT INTO admins ( first_name, last_name, role, email, password)
    VALUES (
            (SELECT first_name FROM data ORDER BY RAND() LIMIT 1),
            (SELECT last_name FROM data ORDER BY RAND() LIMIT 1),
            (SELECT role FROM data ORDER BY RAND() LIMIT 1),
            (CONCAT(MD(UUID()), ‘@TEST.OAMK’),
            (SELECT password FROM data ORDER BY RAND() LIMIT 1)
    );
    SET i = i + 1;
    END WHILE;
    COMMIT;
END
`
````

Figure 12. Code for the procedure to generate data for testing.

6.2 The result compared to the goal.

6.2.1 Employer website

The result of the employer website does allow the user to log in with the admin email and password. After entering the credential, the user will be prompted according to whether the credential is correct or not.

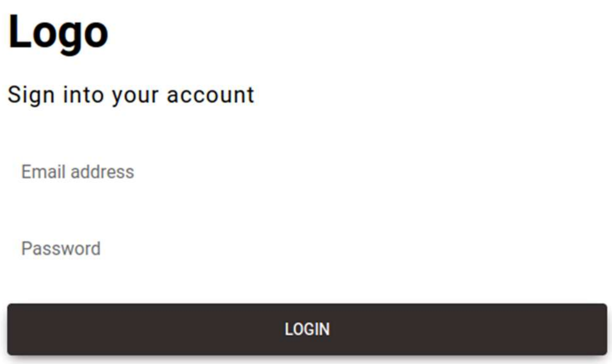
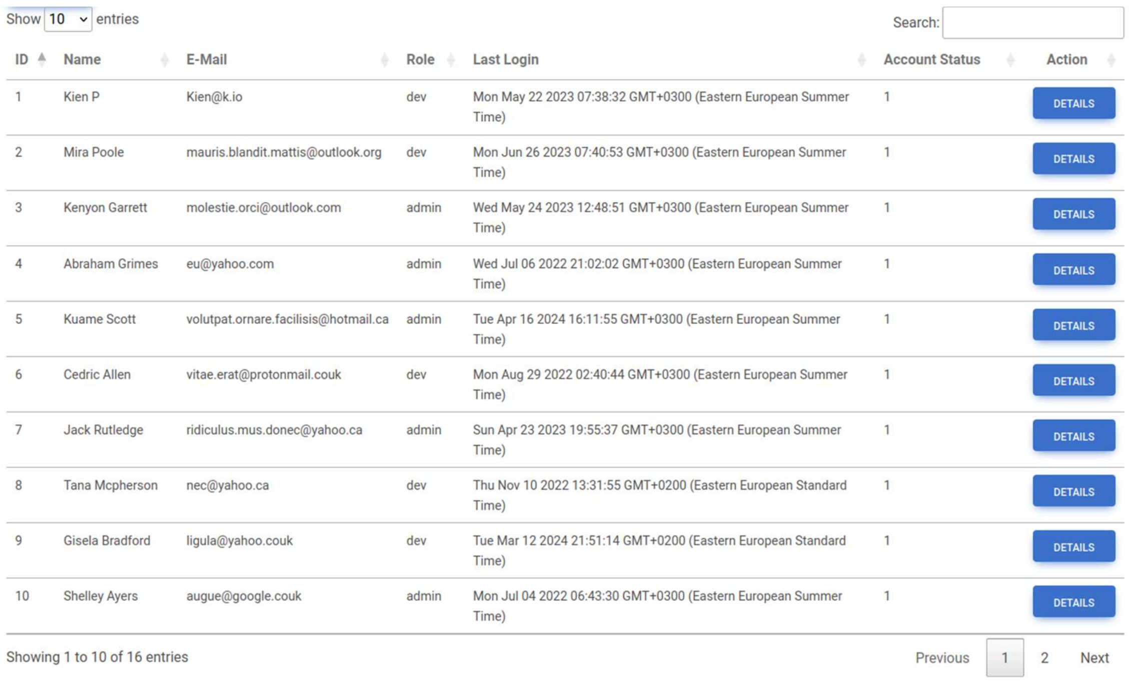


Figure 13. The login page of the employer website.

Upon entering the website, the user can see the list of all the admin, or the employee depending on which tab the user chooses in the navigation bar (Figure 15). There are also forms available for inputting new accounts or changing passwords. In the lists of both admins and employees, the user can sort the table, choose to show the number of entries per page as it is paginated and search for the entry required(Figure 14).



ID	Name	E-Mail	Role	Last Login	Account Status	Action
1	Kien P	Kien@k.io	dev	Mon May 22 2023 07:38:32 GMT+0300 (Eastern European Summer Time)	1	DETAILS
2	Mira Poole	mauris.blandit.mattis@outlook.org	dev	Mon Jun 26 2023 07:40:53 GMT+0300 (Eastern European Summer Time)	1	DETAILS
3	Kenyon Garrett	molestie.orci@outlook.com	admin	Wed May 24 2023 12:48:51 GMT+0300 (Eastern European Summer Time)	1	DETAILS
4	Abraham Grimes	eu@yahoo.com	admin	Wed Jul 06 2022 21:02:02 GMT+0300 (Eastern European Summer Time)	1	DETAILS
5	Kuame Scott	volutpat.ornare.facilisis@hotmail.ca	admin	Tue Apr 16 2024 16:11:55 GMT+0300 (Eastern European Summer Time)	1	DETAILS
6	Cedric Allen	vitae.erat@protonmail.couk	dev	Mon Aug 29 2022 02:40:44 GMT+0300 (Eastern European Summer Time)	1	DETAILS
7	Jack Rutledge	ridiculus.mus.donec@yahoo.ca	admin	Sun Apr 23 2023 19:55:37 GMT+0300 (Eastern European Summer Time)	1	DETAILS
8	Tana Mcpherson	nec@yahoo.ca	dev	Thu Nov 10 2022 13:31:55 GMT+0200 (Eastern European Standard Time)	1	DETAILS
9	Gisela Bradford	ligula@yahoo.couk	dev	Tue Mar 12 2024 21:51:14 GMT+0200 (Eastern European Standard Time)	1	DETAILS
10	Shelley Ayers	augue@google.couk	admin	Mon Jul 04 2022 06:43:30 GMT+0300 (Eastern European Summer Time)	1	DETAILS

Showing 1 to 10 of 16 entries

Previous 1 2 Next

Figure 14. Example of a list of users along with their roles, name, and last login time.

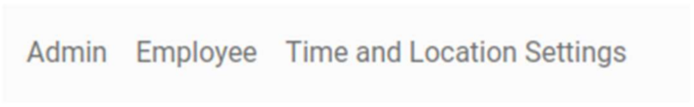


Figure 15. Navigation bar.

In the form to add new admins(Figure 16), the email will need to be unique, or the user will be prompted to enter a new unique email as the email will be used to log in.

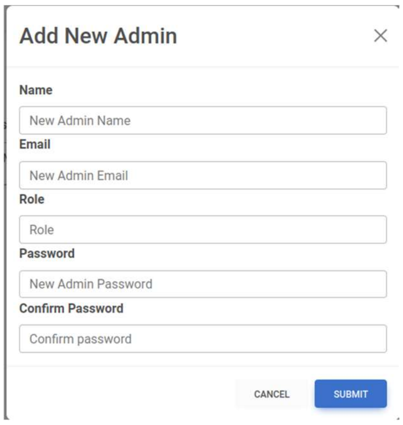
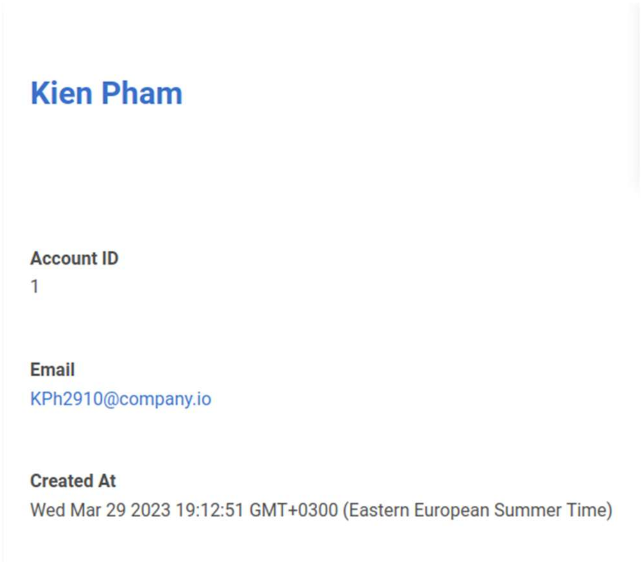


Figure 16. Example of a form.

On each employee profile page, the employer can see their details (Figure 17) and their work time both on per day or monthly basis (Figures 18 and 19).



Kien Pham

Account ID
1

Email
KPh2910@company.io

Created At
Wed Mar 29 2023 19:12:51 GMT+0300 (Eastern European Summer Time)

Figure 17. Employee details.

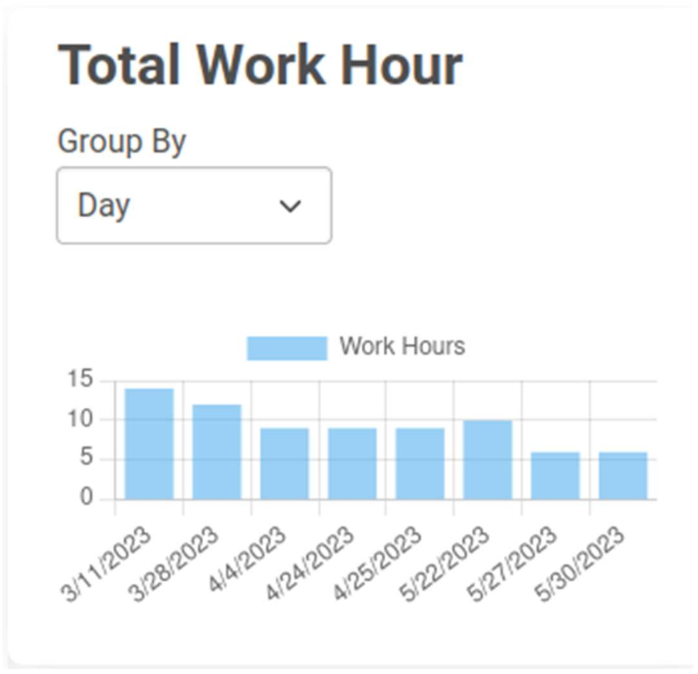


Figure 18. Total work hour chart group by day.

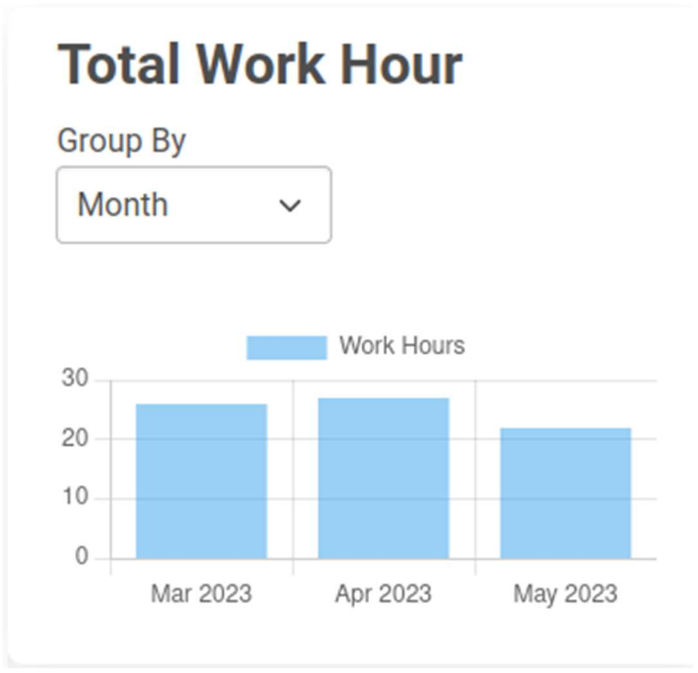


Figure 19. Total work hours sorted by month.

Compared to the set goals, the employer website result had fulfilled all of it from login, seeing all the employee data to sorting the working hour by day and month.

6.2.2 Employee Application

The result application is a working hybrid application using React Native. The screenshot in this section is taken using Expo web view (Figure 21) and an Android simulator (Figure 20) to show that it works both as a web app and natively in an Android environment This application allows the user to log in using their credential (Figure 20).

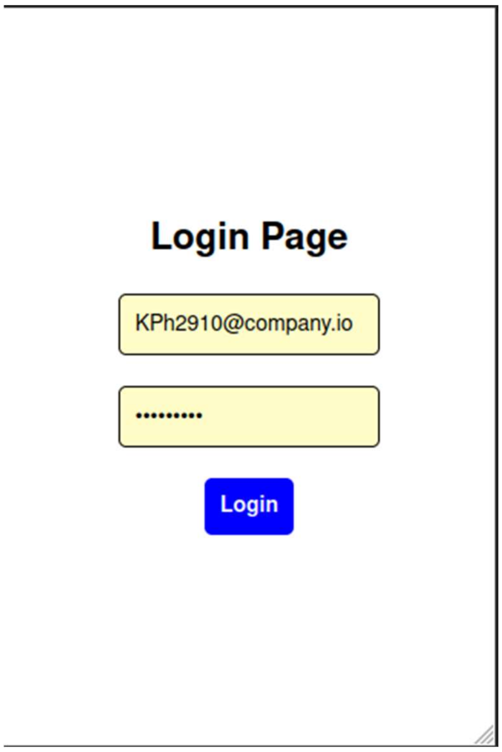


Figure 20. login page of the mobile app.

After login, the user can access the main page which contains a button for check-in, checkout, change password, custom check-in/out and logout which fulfilled the goals for the application set out in the Goal section.

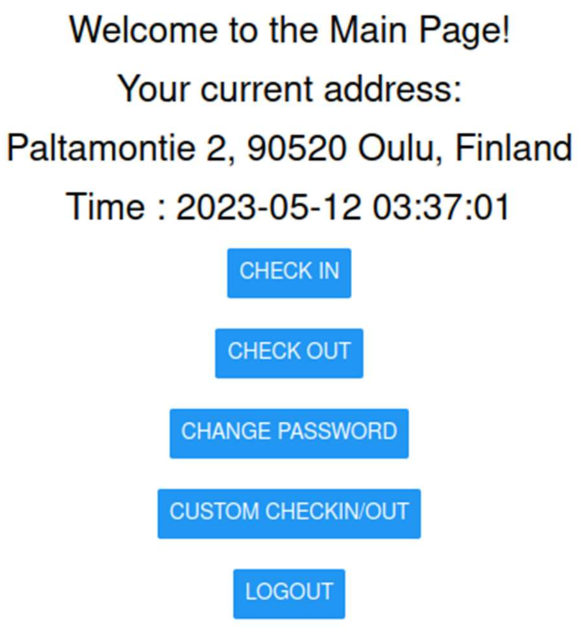

Welcome to the Main Page!
Your current address:
Paltamontie 2, 90520 Oulu, Finland
Time : 2023-05-12 03:37:01

Figure 21. The main page of the employee's application.

7 CONCLUSION

The main objective of this thesis is to create a system in which there are a website, a backend API, and a mobile application for small business to monitor their employee worktime using GPS as a tool to check in and check out of the workplace. During the making of this project, the author learnt more in-depth about React Native, Express JS, MySQL and SQL and JavaScript and API.
The result of the project while having fulfilled the goal set out at the beginning, it still hasn't reached the point where it can be deployed to the production level and be used as an employee management system. The GPS while does result in an accurate address each time, the author had to point out that the location in which the system is tested is only in Oulu, Finland and the premises of the address used for testing is quite large, in a more dense area like in the middle of the city centre where premises of each business is small, the GPS might result in an inaccurate reading, rendering the time recorded inaccurately. The design of the application UI is also still very basic with white background and blue buttons, which are not pleasing to the eyes and can be not very end-user friendly. However, the project is built that new functionality, and the changing of the UI element are easily implemented in the future.
Due to the scope of the project, some areas haven't been monitored such as deploying the application on a public domain using AWS or similar services or testing with scripts. The application had only successfully deployed on a local level which did successful and functional.
The result of this project is a management system that with more refinement can be used in small businesses that required their employee to be in the physical location like small restaurant or small shops.

7.1 Future Improvement

Due to time constrain, testing wasn't done with any testing script which would provide a better and more precise result. In future versions, other than the unfulfilled goals, salary calculation with custom tax, salary for overtime, and insurance should be added for a more complete experience for small businesses which this app aims at. Furthermore, in the working hour chart, more ways of sorting like sorting in a user-set time range would be better for the employer.

REFERENCES

1. About Node.js. Node.js. [Cited: 20 April 2023.] https://nodejs.org/en/about.

2. About npm. npm Docs. [Cited: 20 April 2023.] https://docs.npmjs.com/about-npm.

3. ExpressJS. [Cited: 20 April 2023.] https://expressjs.com/.

4. Oracle. What is MySQL? [Cited: 20 April 2023.]. https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html.

5. Eemisse, Matthew. EJS. EJS. [Cited: 20 April 2023.] https://ejs.co/.

6. Meta OpenSource. React Native. React Native. [Cited: 20 April 2023.] https://reactnative.dev/.

7. Expo. Overvew. Expo docs. [Cited: 20 April 2023.] https://docs.expo.dev/overview/.

8. Google. Google API overview. Google Maps Platform. [Cited: 20 April 2023.] https://developers.google.com/maps/documentation/geocoding/overview.

9. Postman, Inc. About Postman. Postman. [Cited: 20 April 2023.] https://www.postman.com/company/about-postman/.

10. Mozilla Foundation. Basic of HTTP. mdn web docs. [Cited: 20 April 2023.] https://www.postman.com/company/about-postman/.

11. Redis Ltd. Introduction to Redis. Redis. [Cited: 20 April 2023.] https://redis.io/docs/about/.

12. National Geographic Society. GPS. National Geographic. [Cited: 20 April 2023.] https://education.nationalgeographic.org/resource/gps/.

13. Amazon Web Service, Inc. What's The Difference Between Web Apps, Native Apps, And Hybrid Apps? AWS. [Cited: 20 April 2023.] https://aws.amazon.com/compare/the-difference-between-web-apps-native-apps-and-hybrid-apps/.

14. Fatunmbi, Teniola. A Comparison of Cookies and Tokens for Secure Authentication. Okta Developer. 8 Feb 2022. [Cited: 20 April 2023.] https://developer.okta.com/blog/2022/02/08/cookies-vs-tokens.

15. Madurai, Vivek. Different ways to Authenticate a Web Application. Medium. 5 Feb 2018. [Cited: 20 April 2023.] https://medium.com/@vivekmadurai/different-ways-to-authenticate-a-web-application-e8f3875c254a.

APPENDIX

```
CREATE TABLE `admins` (
  `id` int PRIMARY KEY NOT NULL AUTO_INCREMENT,
  `email` varchar(128) NOT NULL DEFAULT "",
  `password` varchar(64) DEFAULT "",
  `name` varchar(128) NOT NULL DEFAULT "",
  `role` varchar(128) DEFAULT NULL,
  `created` datetime NOT NULL DEFAULT (CURRENT_TIMESTAMP),
  `last_login` datetime DEFAULT NULL,
  `is_active` tinyint NOT NULL DEFAULT "1"
);
CREATE TABLE `employee` (
  `id` int PRIMARY KEY NOT NULL AUTO_INCREMENT,
  `email` varchar(128) NOT NULL DEFAULT "",
  `password` varchar(64) DEFAULT "",
  `name` varchar(128) NOT NULL DEFAULT "",
  `created` datetime NOT NULL DEFAULT (CURRENT_TIMESTAMP),
  `last_login` datetime DEFAULT NULL,
  `is_active` tinyint NOT NULL DEFAULT "1"
);

CREATE TABLE `employee_checkin_time` (
  `id` int PRIMARY KEY NOT NULL AUTO_INCREMENT,
  `type` ENUM ('in', 'out') NOT NULL,
  `time` datetime NOT NULL,
  `employee_id` int NOT NULL,
  `custom_time` tinyint NOT NULL DEFAULT "0"
);

CREATE TABLE `settings` (
  `id` int NOT NULL AUTO_INCREMENT,
  `working_time` json DEFAULT NULL,
  `location` varchar(256) DEFAULT NULL
);

CREATE UNIQUE INDEX `email` ON `admins` (`email`);

CREATE UNIQUE INDEX `email` ON `employee` (`email`);

ALTER TABLE `employee` ADD FOREIGN KEY (`id`) REFERENCES
`employee_checkin_time` (`employee_id`);
```

Appendix 1. Database table create command.

Appendix 2. API documentation.

'/auth/login': Get login cookie to access admins and time route.

POST

- Request:
  - Body: JSON

    {

    email : string,

    password : string

    }
- Response:
  - If credential correct:
    - Return cookies and Raw/JSON in body :

      { success : 'logged in!' }
  - If credential incorrect:
    - Return errors.

'/login'

GET

- Request: Doesn't has any requirement.
- Response:
  - Return login.EJS

/

GET

- Request:
  - Header:
    - Include cookies received after **/auth/login.**
- Response:
  - If correct cookies:
    - Return 'index.EJS'.
  - Incorrect cookies:
    - Redirect to '/login'**.**

'/admin/'

GET: Get admin list page.

- Request:

    - Header:
        - Include cookies received after ‘/auth/login’**.**
- Response:
    - If correct cookies:
        - Return ‘list.EJS’.
    - Incorrect cookies:
        - Redirect to ‘/login’**.**

GET: Get admin page with ID number.

- Request:
    - Header:
        - Include cookies received after ‘/auth/login’**.**
    - Params: ‘/{ID : int}’
- Response:
    - If correct cookies:
        - Return admin data in form of EJS file for display.
    - Incorrect cookies:
        - Redirect to ‘/login’**.**
    - Invalid ID:
        - Redirect to ‘/admin’
        - Return code 401 with message in body: ‘{ errors : ‘Invalid ID’ }’.

‘/admin/new’

POST: Add new admin.

- Request:
    - Header:
        - Include cookies received after ‘/auth/login’**.**
    - Body:

      ```
      {
              name : string,
              password : string,
              role : string,
              email : string
      }
      ```
- Response:
    - If correct cookies:

        - Return code 200 with success message in body.
    - Incorrect cookies:
        - Redirect to '/login**'**
    - Duplicate email or empty:
        - Redirect to /admin
        - Return code 401 with error message in body.

'/admin/password'

POST: change password for admin with ID:

- Request:
    - Header:
        - Include cookies received after '/auth/login'.
    - Body:

      {

          password : string,

          newPassword : string,

      }
- Response:
    - If correct cookies:
        - Return code 200 with success message in body.
    - Incorrect cookies:
        - Redirect to '/login'.
    - Incorrect old password or empty new password:
        - Redirect to '/admin'
        - Return code 401 with error message in body.

‘/employee/’

GET: Get employee list page.

- Request:
    - Header:
        - Include cookies received after ‘/auth/login.’
- Response:
    - If correct cookies:
        - Return ‘list.EJS’
    - Incorrect cookies:
        - Redirect to ‘/login’.

POST: To verify employee credential

- Request:
    - Params : include API key:

      ‘/employee/API_Key’
    - Body:

      ```
      {
              email : string,
              password : string
      }
      ```
- Response:
    - If correct credential:
        - Return code 200 and JSON:

          ```
          {
                  employeeId : int,
                  success : ‘logged in’
          }
          ```
    - Incorrect credential:
        - Return code 401 and error message.

          { errors : Error Message }

GET: Get employee page with ID number.

- Request:
    - Header:
        - Include cookies received after '/auth/login'.
    - Params: '/{ID : int}'
- Response:
    - If correct cookies:
        - Return employee data in form of EJS file for display.
        - Format:

```
{
        ID : int,
        email: string,
        name: name,
        created : datetime,
        timeData: {
                [
                        date : date ( DD/MM/YYYY),
                         workTime : float (round to the first deci-
                         mal place)
                ],
                [
                        …
                ],
                …
        }
}
```

        - timeData are calculated in the server to consolidate workhour to individual day from check-in and check-out data.
    - Incorrect cookies:
        - Redirect to '/login'
    - Invalid ID:
        - Redirect to '/admin'
        - Return code 401 with message in body: { errors : 'Invalid ID' }

‘/employee/time’

POST Send in check in/out request with time and location to check in/out.

- Request:
  - Params : include API key:

    ‘/employee/time/API_Key’
  - Body:

    ```
    {
            ID : int,
            type : string ( accept ‘in’ or ‘out’),
            time : datetime,
            customTime: in (accept 1 or 0),
            location : string
    }
    ```
- Response:
  - If correct API key and correct syntax:
    - Return code 200 and JSON:

      ```
      {
              success : ‘Checked in/out’
      }
      ```
  - Incorrect syntax/ Incorrect Location:
    - Return code 401 and error message.

      ‘{ errors : Error Message }’

'/settings'

GET Get the setting.EJS file.

- Request:
  - Header:
    - Include cookies received after '/auth/login**'.**
- Response:
  - If correct cookies:
    - Return 'settings.EJS'
  - Incorrect cookies:
    - Redirect to '/login'

POST Get the current setting.

- Request:
  - Header:
    - Include cookies received after '/auth/login'.
- Response:
  - If correct cookies:
    - Return code 200 and JSON file containing current settings.

      ```
      {
              workingTime :
                      {
                              in : string ( between '0000' and '2400')
                              out : string ( between '0000' and '2400')
                      },
              location : string
      }
      ```
  - Incorrect cookies:
    - Redirect to '/login'.

‘/settings/update’

POST Update current setting.

- Request:
    - Header:
        - Include cookies received after ‘/auth/login’.
    - Body:
        - Include a JSON file containing the new settings.

```
{
        workingTime :
                {
                        in : string ( between ‘0000’ and ‘2400’)
                        out : string ( between ‘0000’ and ‘2400’)
                },
        location : string
}
```

- Response:
    - If correct cookies and correct format:
        - Return ‘list.EJS’
    - Incorrect cookies:
        - Redirect to ‘/login**’.**
    - Incorrect format:
        - Return code 401 and error message.